Números y aritmética

Arquitectura de Computadores – IIC2343

Cuando se trata de números, las personas pensamos en base 10 (números *decimales*):

- diez símbolos diferentes — 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 — llamados dígitos decimales, o simplemente, dígitos

Pero los números pueden ser representados en cualquier base

… p.ej., en base 2 (números *binarios*):

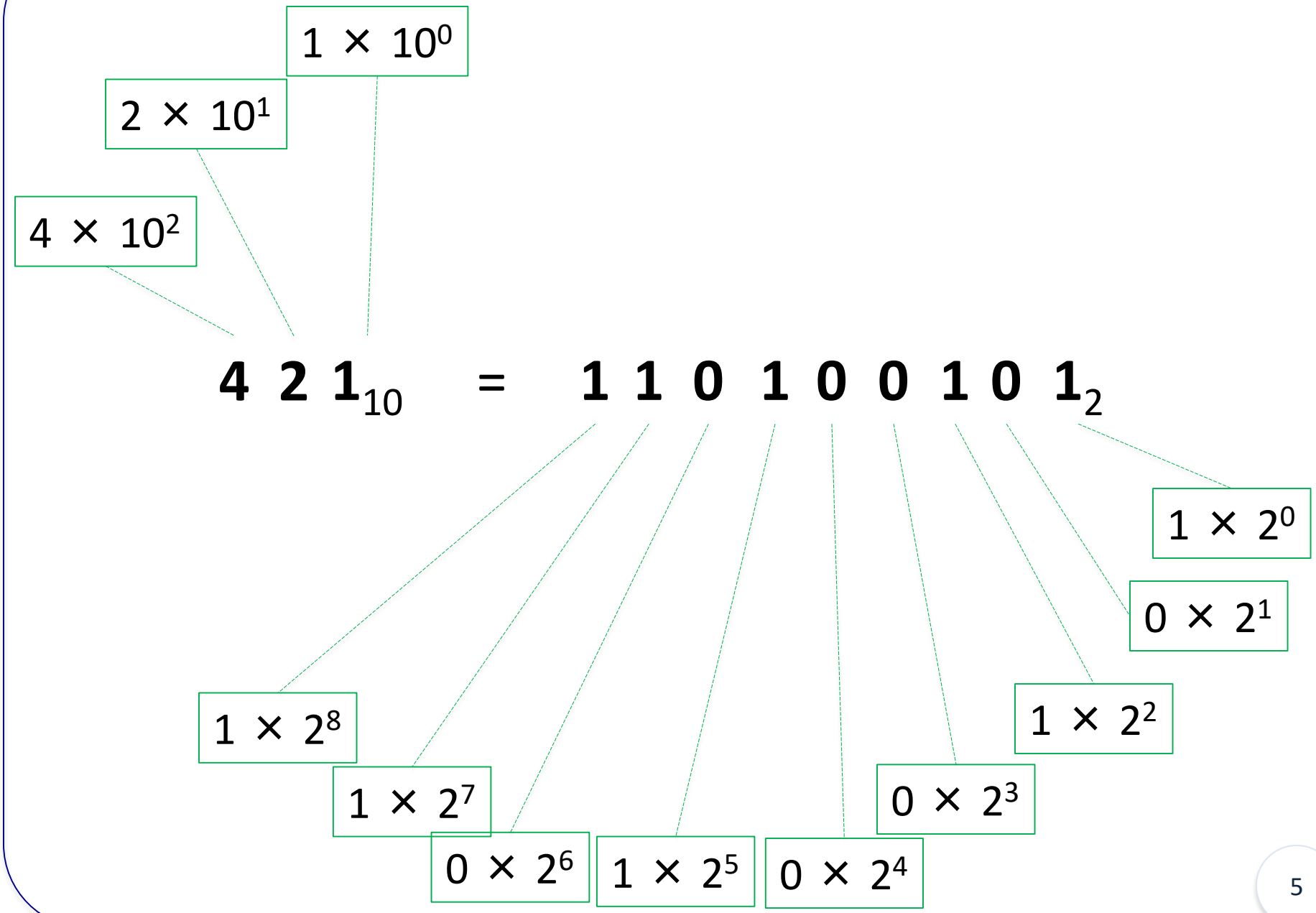
- dos símbolos diferentes — 0, 1 — llamados dígitos binarios o bits

Los bits son los “átomos” de la computación:

- toda información se compone de bits

Numeramos los bits de un número binario 0, 1, 2, 3, … de derecha a izquierda:

- es decir, desde el bit menos significativo al bit más significativo


1 × 10^0
2 × 10^1
4 × 10^2
4 2 1_10 = 1 1 0 1 0 0 1 0 1_2
1 × 2^0
0 × 2^1
1 × 2^2
0 × 2^3
1 × 2^5
0 × 2^4
0 × 2^6
1 × 2^7
1 × 2^8

La cantidad de memoria disponible para almacenar un número queda fija al momento de diseñar el computador:

números de precisión finita

P.ej., el conjunto de enteros positivos representables mediante tres dígitos decimales, sin punto decimal ni signo:

000, 001, 002, …, 999

Es imposible representar ciertos números:

- mayores que 999, negativos, fracciones, irracionales, complejos

El conjunto no es cerrado con respecto a la suma, resta o multiplicación:

- 600 + 600 = 1200 → muy grande
- 050 x 050 = 2500 → muy grande

003 – 005 = -2 → negativo

007 / 002 = 3.5 → no es un entero

El álgebra de los números de precisión finita es diferente del álgebra normal:

- p.ej., la ley de asociatividad $a + (b - c) = (a + b) - c$ no se cumple si $a = 700$, $b = 400$ y $c = 300$, porque al calcular $a + b$ en el lado derecho produce *overflow*

Obviamente, no es que los computadores sea especialmente inadecuados para hacer aritmética, sino que

es importante entender cómo funcionan

También tenemos que representar números negativos

Podemos agregar un signo, representado en un bit: ***signo y magnitud***

Problemas:

- este bit, ¿va a la derecha o a la izquierda?
- al sumar, se necesita un paso adicional para saber el valor de este bit
- hay un cero positivo y un cero negativo

No habiendo una mejor alternativa obvia

… se eligió la representación que hacía simple el hardware —***complemento de 2***:

- 0s a la izquierda significa positivo
- 1s a la izquierda significa negativo

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2 = 0_{10}$

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2 = 1_{10}$

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_2 = 2_{10}$

...

$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_2 = 2{,}147{,}483{,}645_{10}$

$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2 = 2{,}147{,}483{,}646_{10}$

$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2 = 2{,}147{,}483{,}647_{10}$

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2 = -2{,}147{,}483{,}648_{10}$

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2 = -2{,}147{,}483{,}647_{10}$

$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_2 = -2{,}147{,}483{,}646_{10}$

...

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_2 = -3_{10}$

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2 = -2_{10}$

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2 = -1_{10}$

En 32 bits (diapositiva anterior):

- la mitad positiva, de 0 a $2{,}147{,}483{,}647_{10}$ ($= 2^{31} - 1$) usa la misma representación que antes
- el patrón de bits que sigue ($1000\ldots0000_2$) representa el número más negativo $-2{,}147{,}483{,}648_{10}$ ($= -2^{31}$)

  … el cual no tiene un número positivo correspondiente
- y luego viene una secuencia de números negativos de magnitud decreciente, desde $-2{,}147{,}483{,}647_{10}$ ($= 1000\ldots001_2$) hasta -1_{10} ($= 1111\ldots1111_2$)

Todo computador hoy en día usa complemento de 2 para representar números con signo (*signed numbers*)

string	unsigned	sign magnitude	1's complement	2's complement
0000	0	0	0	**0**
0001	1	1	1	**1**
0010	2	2	2	**2**
0011	3	3	3	**3**
0100	4	4	4	**4**
0101	5	5	5	**5**
0110	6	6	6	**6**
0111	7	7	7	**7**
1000	8	–0	–7	**–8**
1001	9	–1	–6	**–7**
1010	10	–2	–5	**–6**
1011	11	–3	–4	**–5**
1100	12	–4	–3	**–4**
1101	13	–5	–2	**–3**
1110	14	–6	–1	**–2**
1111	15	–7	–0	**–1**

Todos los números negativos tienen un 1 en el bit más significativo:

- *bit de signo*
- basta examinar este bit para saber si un número es positivo o negativo (0 se considera positivo)

Así, con 32 bits, el valor del número representado es

$$(x_{31} \times -2^{31}) + (x_{30} \times 2^{30}) + (x_{29} \times 2^{29}) + \ldots + (x_1 \times 2^1) + (x_0 \times 2^0)$$

Ocurre *overflow* cuando el resultado de la operación produce un bit de signo incorrecto:

- un 0 a la izquierda cuando el número es negativo
- un 1 a la izquierda cuando el número es positivo

Cómo determinar el inverso aditivo de un número binario en complemento de 2

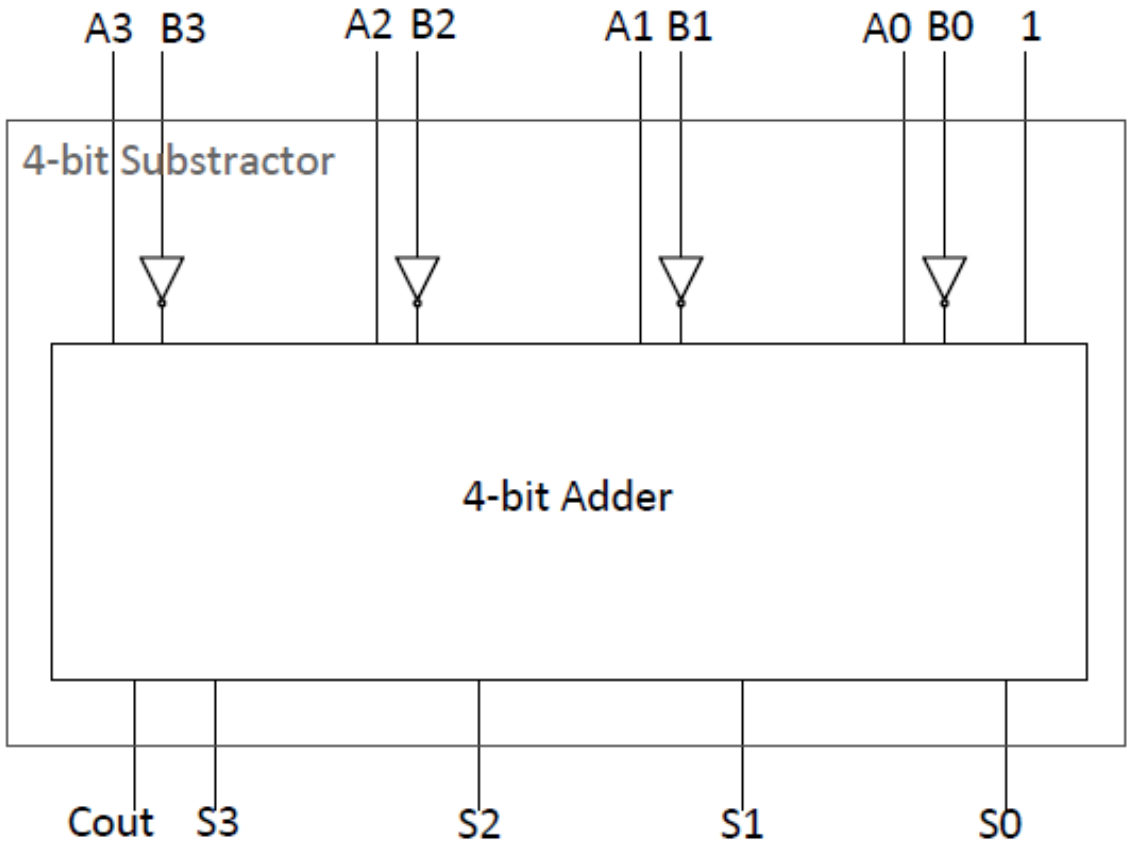
Cómo convertir un número binario representado en *n* bits a un número binario representado con más de *n* bits

La clave consiste en tener módulos combinables y expandibles

¿Qué debemos hacer para diseñar un restador?

Afortunadamente, no necesitamos mucho más.

Aprovechando el complemento de 2, sumamos al minuendo el inverso aditivo del sustraendo y además ponemos en 1 el carry-in.


A3 B3
A2 B2
A1 B1
A0 B0
1
4-bit Substractor
4-bit Adder
Cout
S3
S2
S1
S0

¿Podemos hacer algo con esto?

- Nuestro primer paso será un sumador/restador de 4 bits
- ¿Podemos construir esto usando sólo los elementos vistos hasta ahora en clases?

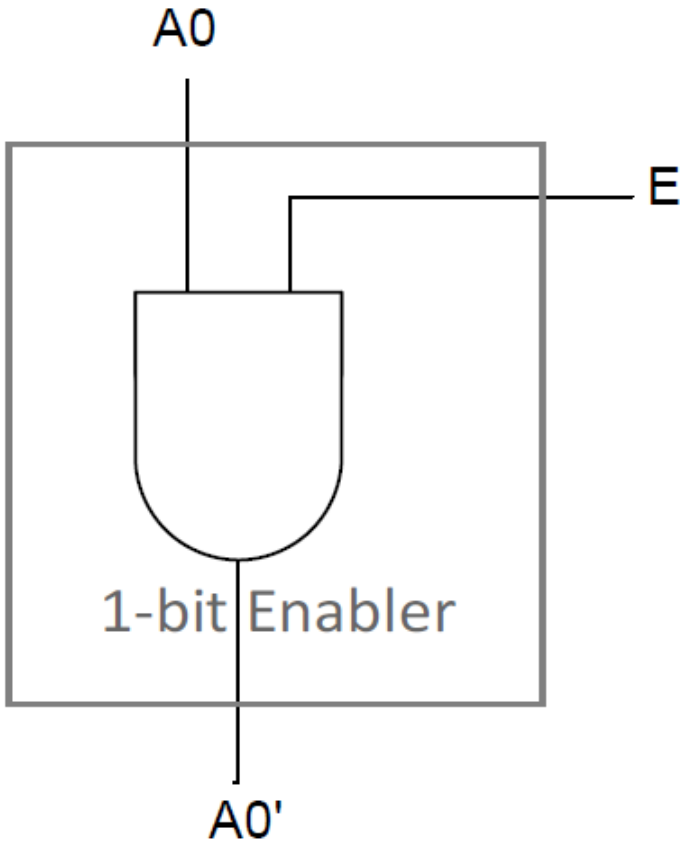
Elementos de control son necesarios

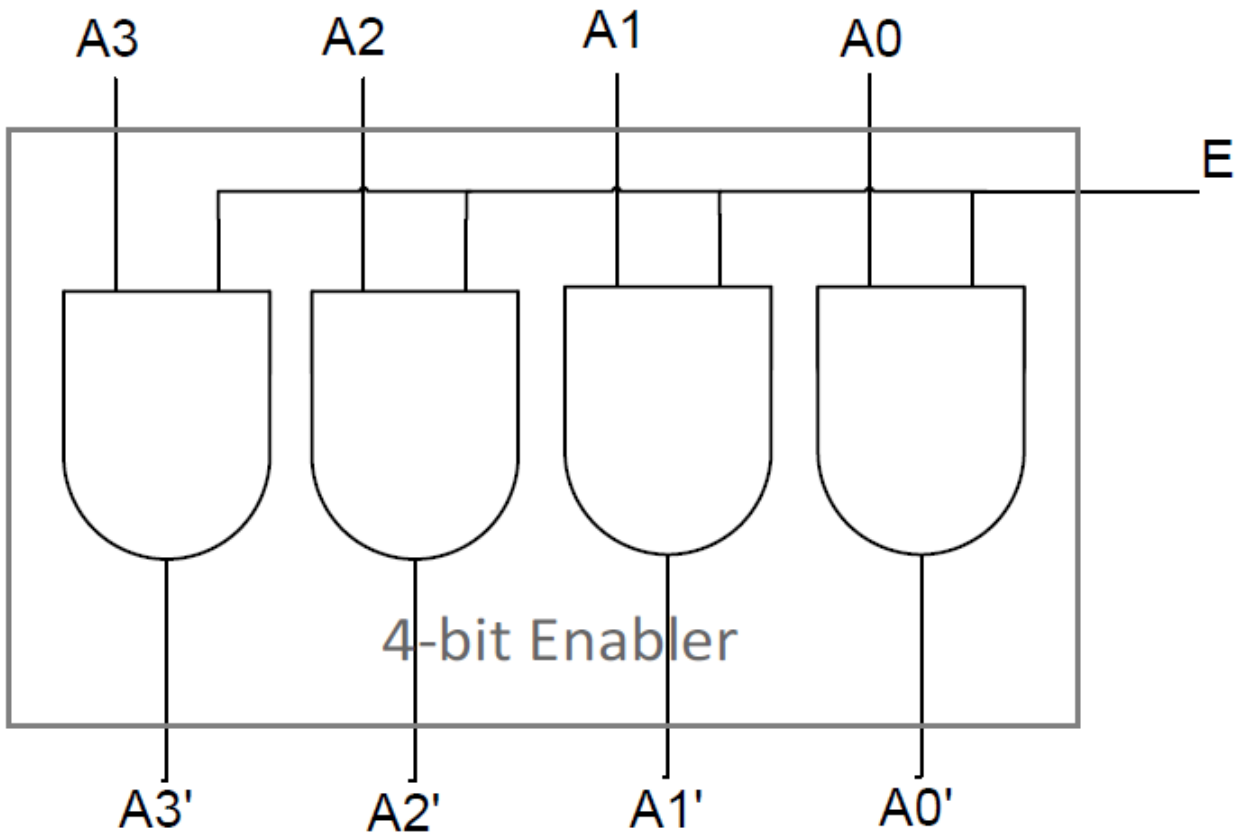
- Para diseñar un el sumador/restador aún nos faltan algunos elementos lógicos.
- Estos tienen relación con el control y la selección de las salidas y operaciones que se realizan.
- Estos elementos se conocen como Enablers y Multiplexores o Mux.

Buses otorgan simplificación en el diseño de los componentes

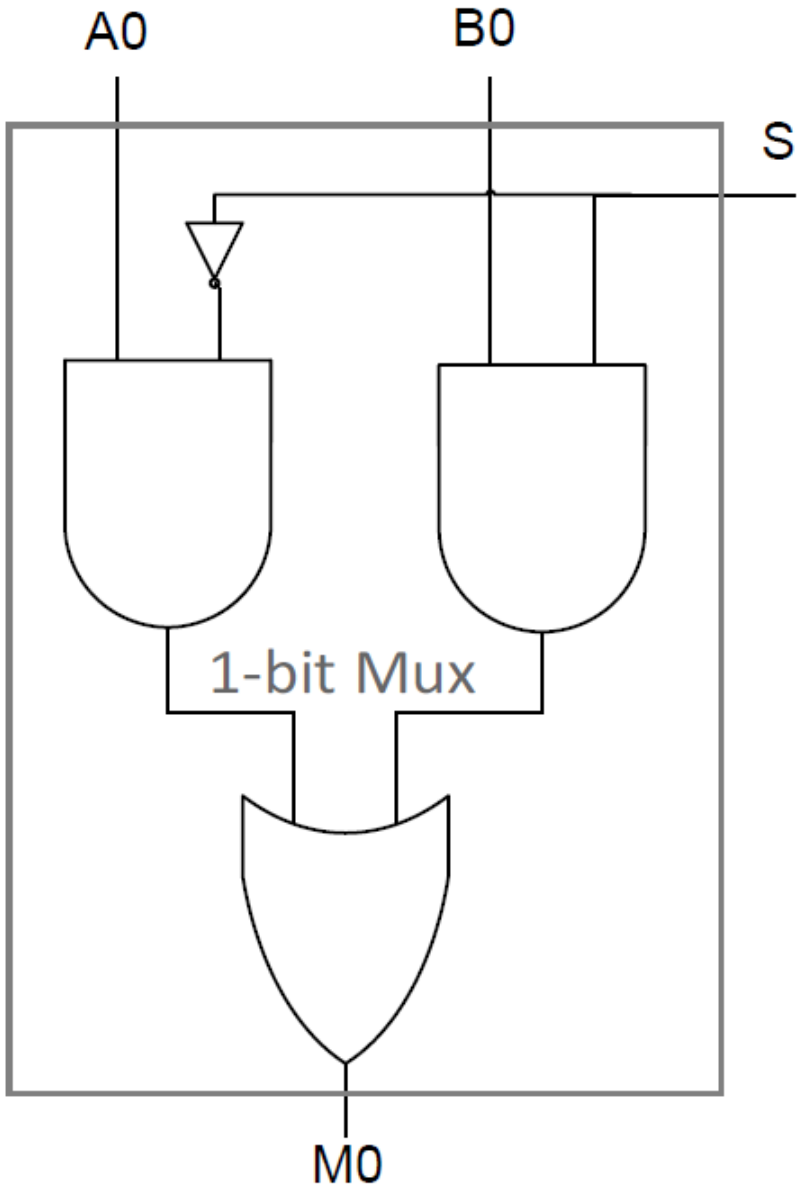
- Las conexiones usadas anteriormente sólo transmitían datos, ej. sumador.
- Ahora, además de eso, necesitamos transmitir órdenes de control.
- Así, se hace la distinción entre buses de datos y buses de control.

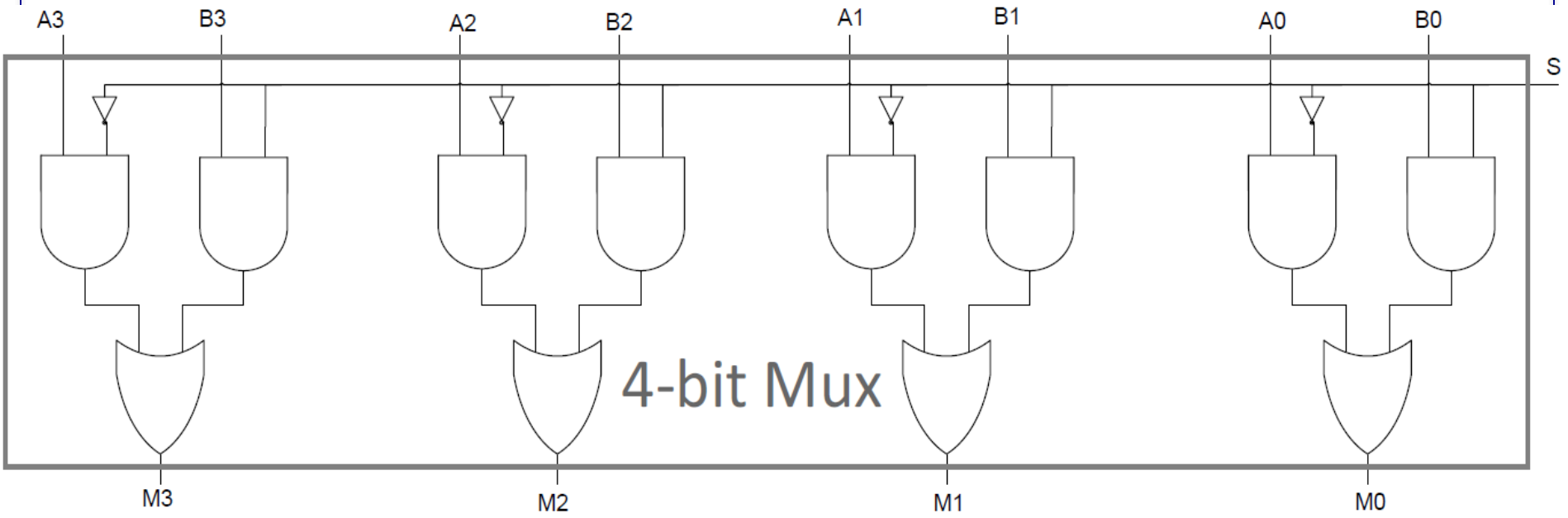
E	A0’
0	0
1	A0



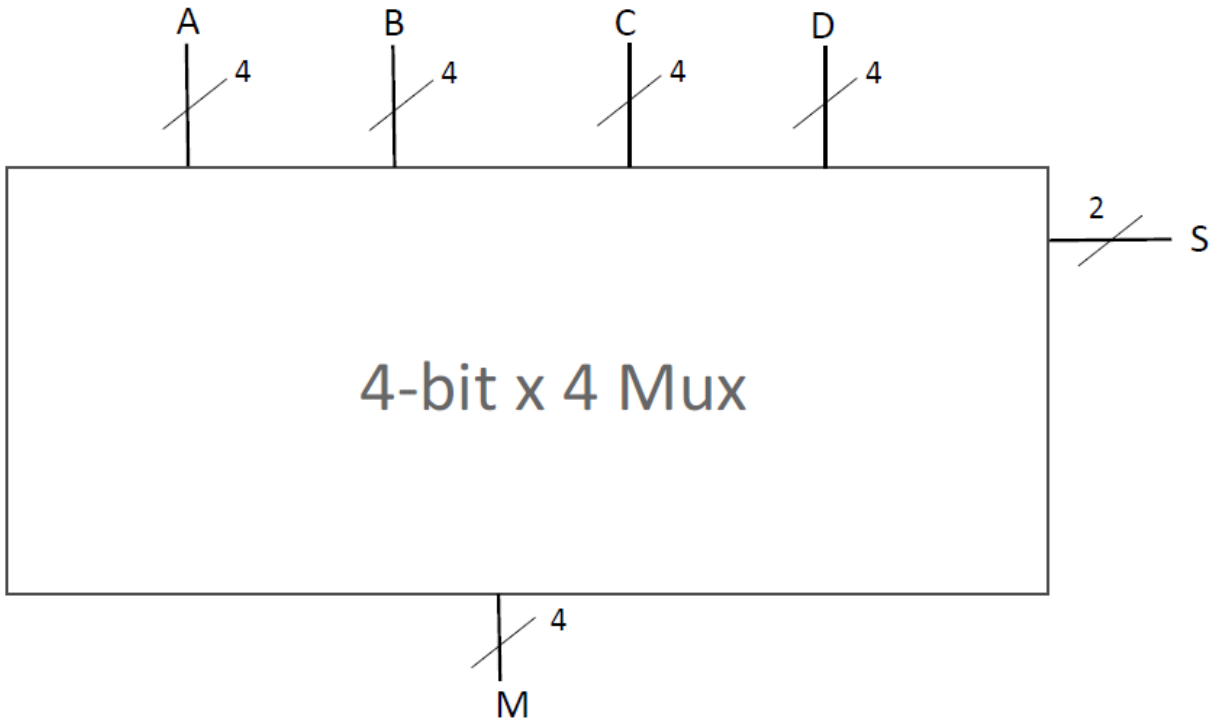

A3
A2
A1
A0
E
4-bit Enabler
A3'
A2'
A1'
A0'

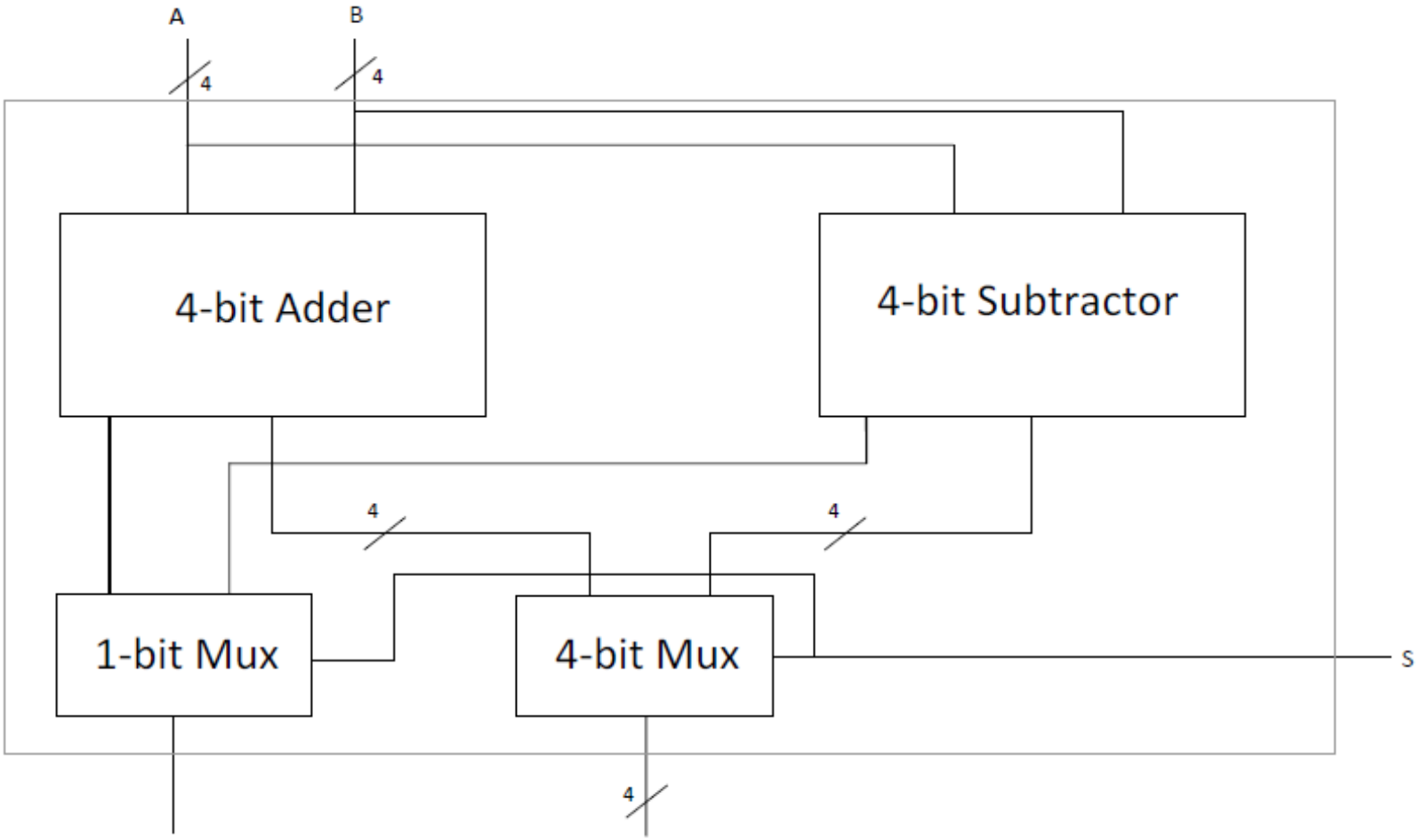
S	M0
0	A0
1	B0

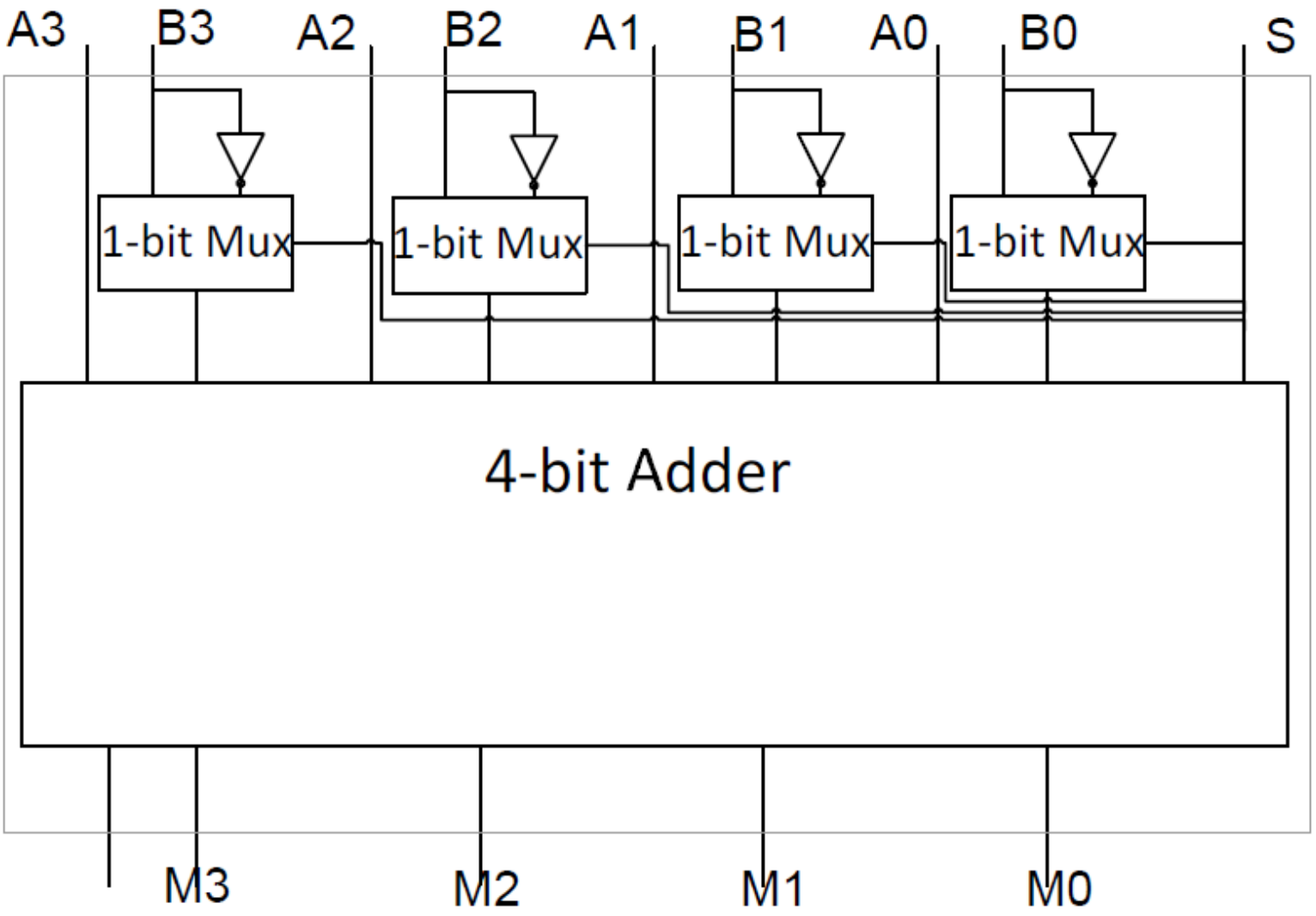



A3
B3
A2
B2
A1
B1
A0
B0
S
4-bit Mux
M3
M2
M1
M0

S1	S1	M
0	0	A
0	1	B
1	0	C
1	1	D




A
B
4
4
4-bit Adder
4-bit Subtractor
4
4
1-bit Mux
4-bit Mux
S
4


A3
B3
A2
B2
A1
B1
A0
B0
S
1-bit Mux
1-bit Mux
1-bit Mux
1-bit Mux
4-bit Adder
M3
M2
M1
M0